NPS52-84-026

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THE FOUR FORMS OF Ω

BRUCE J. MAC LENNAN

DECEMBER 1984

Approved for public release; distribution unlimited

Prepared for: Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

The work reported herein was supported by Contract N00014-85-24057 from the Office of Naval Research.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-84-026	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE FOUR FORMS OF Ω		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) BRUCE J. MAC LENNAN		8. CONTRACT OR GRANT NUMBER(s) N00014-85-24057
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE December 1984
		13. NUMBER OF PAGES 30
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Object-oriented programming, production rules, production systems, concrete syntax, two-dimensional language, pseudo-natural language, knowledge representation, natural language interface, logic programming, simulation languages, knowledge base, office automation, rule-based systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We describe four alternative syntactic forms for the object-oriented, rule-based language Ω. These notations are all different concrete representations of the same abstract language. The first notation uses a predicate logic style. The second has a stylized natural language format. The third extends the second by providing anaphoric reference. The fourth form drops the linear syntax of the first three in favor of a two-dimensional format based on the idea of a form.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

THE FOUR FORMS OF Ω

Alternate Syntactic Forms for an Object-Oriented Language

Bruce J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93943

1. Introduction

In this report[1] we describe four alternative syntactic forms for the object-oriented language Ω, described in our *View of Object-Oriented Programming* (Naval Postgraduate School Computer Science Dept. Tech. Rept. NPS52-83-001, Feb. 1983). Additional information on a prototype implementation of Ω can be found in Heinz M. McArthur's *Design and Implementation of an Object-Oriented, Production-Rule Interpreter* (Naval Postgraduate School Master's Thesis, Dec. 1984).

It must be emphasized that these notations are all different concrete representations for the same abstract language. Thus, for example, rules could be entered in one form and displayed in another. This permits different users (or the same user at different points in time) to look at a program in different ways.

The first syntactic form, Ω_1, uses a predicate logic style. It is also the simplest to parse. The second and third styles (Ω_2 and Ω_3) have a stylized natural language format. As a result they are less compact, but more readable to computer-naive users. The fourth form, Ω_4, drops the linear syntax of the other three, and adopts a two-dimensional format based on the idea of a form. It is the least compact, but most amenable to use by computer-naive users.

We describe each of these forms in the body of this report, and illustrate the forms with a simple example. A formal grammar for each format appears in the Appendices.

1. Work reported herein was supported in part by the Office of Naval Research under contract number N00014-85-24057.

2. First Form

The first form, Ω_1 is based on a predicate-logic style notation. It will look familiar to readers acquainted with logic programming languages and production rule systems. The basic construct is the *rule,* which has the form

$$\textit{cause} \Rightarrow \textit{effect}$$

The *cause* describes a possible situation in a space of objects connected by relations. If that situation holds, then the rule may be applied, which means that the actions described by its *effect* part will be performed. Rules are executed *indivisibly,* which means that it is guaranteed that the situation still holds when the actions are performed.

There is normally no order implied between rules; they can be tested in any order. However, rules can be connected by the word else when a particular order must be imposed:

$$\begin{array}{rl} & \textit{cause}_1 \Rightarrow \textit{effect}_1 \\ \text{else} & \textit{cause}_2 \Rightarrow \textit{effect}_2 \\ & \vdots \\ \text{else} & \textit{cause}_n \Rightarrow \textit{effect}_n \end{array}$$

In this case, the second and succeeding rules are tried only when the preceding rules have failed.

The cause part of a rule describes a situation in terms of one or more *conditions,* which represent the presence or absence of relationships between objects:

$$\textit{condition}_1, \textit{condition}_2, \ldots, \textit{condition}_n$$

All of these conditions must be satisfied before a rule can be applied.

A condition for testing for the *presence* of a tuple in a relationship has the form:

$$\textit{primary} \ (\textit{pattern}_1, \textit{pattern}_2, \ldots, \textit{pattern}_n)$$

The *primary* is an expression (usually just an identifier) that evaluates to a relation. The following list of patterns defines a *tuple pattern.* Each pattern in the list can be either an free (i.e., undefined) variable, or an expression containing no free (i.e., only bound) variables. An expression is evaluated during the matching process, and matches a value equal to the result of its evaluation. A free variable will match any value, but becomes bound to that value during the matching process. The special free variable '—' can be used to match anything without binding a variable name.

A condition for testing for the *absence* of a tuple from a relationship has the form:

$$\neg \textit{primary} \ (\textit{pattern}_1, \textit{pattern}_2, \ldots, \textit{pattern}_n)$$

This condition succeeds only if there is *not* a tuple of the specified form in the relation that is the value of the primary.

Finally, as a convenience we permit *cancel* conditions:

$$^*\textit{primary} \ (\textit{pattern}_1, \textit{pattern}_2, \ldots, \textit{pattern}_n)$$

This tests for the presence of a tuple of the specified form, just as the first kind of condition, but it has a side effect of deleting that tuple if the rule is applied. Although, this could be programmed explicitly, the situation is common enough that it is important to reflect it in the notation.

The effect part of a rule is composed of a sequence of *transactions:*

$$\textit{transaction}_1, \textit{transaction}_2, \ldots, \textit{transaction}_n$$

These transactions can be performed in any order or in parallel. The transactions are of four kinds: assertions, denials, calls and sequential blocks.

An *assertion* is a transaction of the form:

$$\textit{primary} \ (\textit{expression}_1, \textit{expression}_2, \ldots, \textit{expression}_n)$$

Its effect is to add to the relation that is the value of the primary the tuple $< V_1, V_2, \ldots, V_n >$, where each V_i is the value of *expression*$_i$. Typically these expressions contain variables that were bound in the cause part of the rule.

A *denial* has the form of an assertion preceded by a negation sign:

$$\neg \textit{primary} \ (\textit{expression}_1, \textit{expression}_2, \ldots, \textit{expression}_n)$$

Its effect is to *delete* the specified tuple from the specified relation. If this relation does not contain this tuple, then an error condition holds.

A *call* is a transaction of the form:

$$primary\ \{\ expression_1,\ expression_2,\ \ldots,\ expression_n\ \}$$

Its purpose is a form of synchronous communication performed by sending a message through one relation, and waiting for a reply to be returned through another relation. For example, the call

$$P\{E, F\}$$

has the effect of performing the assertion

$$P(a, E, F)$$

Here a is a newly generated relation that will be used for receiving the reply. This assertion presumably requests some actions to be performed by other rules (which are watching P). When the actions are completed, an acknowledgment or reply will be placed in the a relation, which permits the calling rule to complete. Note that rules containing calls in their effect parts are not considered indivisible.

The last kind of transaction is a sequential block, which has the form:

$$\{\ statement_1;\ statement_2;\ \cdots;\ statement_n\ \}$$

The effect of this construct is to execute the component statements in order. A statement is simply a rule, simple or compound, with the additional characteristic that its cause part (and the $\Rightarrow$) can be omitted. This reflects the fact that in a sequential block the performance of actions may be conditioned solely on the performance of the preceding statements.

A user normally interacts with an Ω system by typing statements. Thus the form of an Ω terminal session is:

$statement_1$;

$statement_2$;

$\vdots$

$statement_n$;

Many of the statements typed interactively are isolated effects containing a single call. For example, to define 'Contents' to be a new private relation, a user would enter:

```
Define{ Private, "Contents", NewRel{} };
```

Finally, we need a means for manipulating groups of rules as a unit; this is the *rule denotation* and has the form:

$$\ll compound-rule_1.\ compound-rule_2.\ \cdots\ compound-rule_n.\ \gg$$

Notice that each compound rule is terminated by a period. (A compound rule is simply a rule that may contain elses.)

There follows an Ω_1 session to declare an abstract type manager for stacks. The first group of commands defines the relations that characterize stacks. Next comes a rule denotation containing the rules for managing stacks. Finally, a group of definitions make certain of the relations public, but with restricted capabilities.

STACKS IN Ω_1

```
Define {Private, "Contents", NewRel{}};

Define {Private, "Push", NewRel{}};

Define {Private, "Pop", NewRel{}};

Define {Private, "Destroy", NewRel{}};

Define {Private, "NewStack", NewRel{}};


Define {Private, "Rules",

 <<*Push(A ,X ,S ), *Contents(Y ,S )  =>  Receives(A ,S ), Contents(cons[X , Y ], S ).

   *Pop(A ,S ), *Contents(X ,S )  =>  Receives(A ,first[X ]), Contents(rest[X ],S ).

   *NewStack(A ), *Avail(S )  =>  Receives(A ,S ), Contents(nil,S ).

   *Destroy(A ,S ), *Contents(X ,S )  =>  Receives(A ,X ). >> };

Activate {Rules};


Define {Public, "Push", AddOnly{Push}};

Define {Public, "Pop", AddOnly{Pop}};

Define {Public, "Destroy", AddOnly{Destroy}};

Define {Public, "NewStack", AddOnly{NewStack}}.
```

3. Second Form

The second form of Ω attempts to achieve a more natural notation by permitting relations to be named by templates. For example, we can denote the Contents relation by the template:

– is contents of –

Then, instead of using the notation 'Contents(X ,S)', we can write 'X is contents of S '. Using the more mnemonic 'list' and 'stack' in place of 'X ' and 'S ' yields 'list is contents of stack'. Finally, Ω_2 promotes readability by allowing the use of "noise words":

a list is the contents of the stack

Here, 'a' and 'the' are noise words inserted to improve the continuity of the clause.

Relations that are intended to be called as procedures should have 'does' as the first word in their template. For example, the template for the Push relation is:

— does push — on —

Inquiries and assertions to this relation are made in the usual way, by filling in the blanks:

the agent does push the thing on the stack

However, the relation can be called synchronously by omitting the first argument and the word 'does' (thus converting the declarative into an imperative):

push the thing on the stack

This is analogous to the Ω_1 notation

Push {thing, stack}

which is equivalent to

Push (agent, thing, stack)

Templates that do not begin with 'does' cannot be called as procedures.

An inquiry or assertion is negated by placing the word 'not' after the first word in the template, for example,

the list is not the contents of the stack

The same rule applies to 'does' templates:

the agent does not push the thing on the stack

The structure of rules in Ω_2 is the same as in Ω_1, except that all rules are preceded by 'When' and the word 'then' replaces the arrow '$\Rightarrow$'. The Ω_1 cancellation symbol, '*', is replaced by the word 'given' in Ω_2. In other respects the syntax of Ω_2 closely follows that of Ω_1.

STACKS IN Ω_2

```
Define private name "– is contents of –" to be make new relation;

Define private name "– does push – on –" to be make new relation;

Define private name "– does pop –" to be make new relation;

Define private name "– does request a new stack" to be make new relation;

Define private name "– does destroy –" to be make new relation;


Define private name "Stack Rules" to be

  the rules

    When given an agent does push a thing on a stack

      and given a list is the contents of the stack

    then the agent does receive the stack

      and the appending of the thing and the list

        is the contents of the stack.


    When given an agent does pop a stack

      and given a list is the contents of the stack

    then the agent does receive the first element of the list

      and the rest of the list is the contents of the stack.


    When given an agent does request a new stack

      and given a thing is available

    then the agent does receive the thing

      and nil is the contents of the thing.


    When given an agent does destroy a stack

      and a list is the contents of the stack

    then the agent does receive the list.


  end rules;
```

```
Activate the Stack Rules;
Define public name "– does push – on –"
  to be an add only version of "– does push – on –";
Define public name "– does pop –"
  to be an add only version of "– does pop –";
Define public name "– does request a new stack"
  to be an add only version of "– does request a new stack";
Define public name "– does destroy –"
  to be an add only version of "– does destroy –".
```

4. Third Form

The Ω_3 syntax makes an additional step in the direction of a more natural notation: the provision of *anaphoric reference*. To explain this we need some grammatical terminology. First, phrases which denote a value or object, such as

a list

a brother of Joe

the owner of the file

something which is moving

that which receives the result

are called *noun phrases*. Second, phrases that describe a state, condition or relation, and normally stand after a form of the verb *to be,* such as

hot

less than 100

between 20 and 50

are called *adjective phrases*. Finally, phrases that describe a state, condition, relation or action, and either do not contain a form of *to be,* such as moves

does pop the stack

does not push the object on the stack

connects the terminal to the processor

or contain a form of *to be* followed by a noun or adjective phrase, such as

is a list

is less than 100

is not the brother of Joe

is something which is moving

are called *verb phrases*.

A few simple examples will illustrate the idea of anaphoric reference. Suppose that we have the following inquiries in the cause part of a rule:

T is a terminal and T is available

Anaphoric reference permits this to be written

a terminal is available

The *indefinite determiner* 'a' before the noun 'terminal' implies an inquiry of the form 'T is a terminal'. Furthermore, the use of the phrase 'a terminal' in the clause 'a terminal is available' implies that it is the same terminal T that is available. In essence use of the phrase 'a terminal' implies the existence of an object or value X having the property 'X is a terminal'.

More specifically, the indefinite determiners 'a' and 'an' before a noun N are equivalent to X and generate an inquiry

X is N

Thus, the clause 'an N VP', where VP is a verb phrase, reduces to the two clauses 'X is N and X VP'. The same rule applies even if the noun is followed by arguments. For example, the inquiries

X is a brother of John and X is moving can be written

a brother of John is moving

Note that anaphoric reference requires Ω_3 to distinguish between nouns, adjectives and verbs.

We turn to a more complicated example. Suppose we have the inquiries:

T is a terminal and T is available and T is not broken

Anaphoric reference permits this to be written:

a terminal is available and the terminal is not broken

The use of the *definite determiner* 'the' in the phrase 'the terminal' guarantees that the terminal in question is the same one referred to earlier in the rule.

Definite determiners are also permitted in the effect parts of rules. For example, the rule

When T is a terminal and given T is available then T is allocated.

can be written

When given a terminal is available then the terminal is allocated

The phrase 'the terminal' in the effect part refers to the same terminal mentioned in the cause part.

Finally, Ω_3 provides a limited ability for subordination. For example, the inquiries

X connects the terminal to the processor and X is not busy

can be written

something which connects the terminal to the processor is not busy

In general, if VP is a verb phrase, then the clause 'something which VP' is equivalent to X and generates an inquiry of the form 'X VP'. In other words, the clause

something which VP VP′

reduces to the two clauses

X VP and X VP′

Similarly, the phrase 'that which VP' refers back to something X in a previous inquiry of the form 'X VP'. Indeed, the phrase 'a/an NP' can be considered an abbreviated form of 'something which is NP', and 'the NP' can be considered an abbreviated form of 'that which is NP'.

Another permitted form of subordination is illustrated by the following example. The inquiries

a channel is connected to a device and the device is not busy

can be written

a channel is connected to a device which is not busy

In general the phrase 'NP which VP' is equivalent to NP, but generates the additional inquiry 'NP VP'. The use of anaphoric reference and subordination eliminates almost entirely the need for variable names in rules.

The stack example is almost identical in Ω_2 and Ω_3:

STACKS IN Ω_3

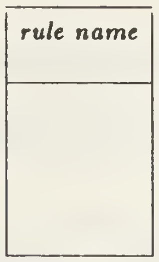
```
Define private noun "contents of –" to be make new relation;
Define private verb "does push – on –" to be make new relation;
Define private verb "does pop –" to be make new relation;
Define private verb "does request new stack" to be make new relation;
Define private verb "does destroy –" to be make new relation;

... remainder as in Ω2 ...
```

5. Fourth Form

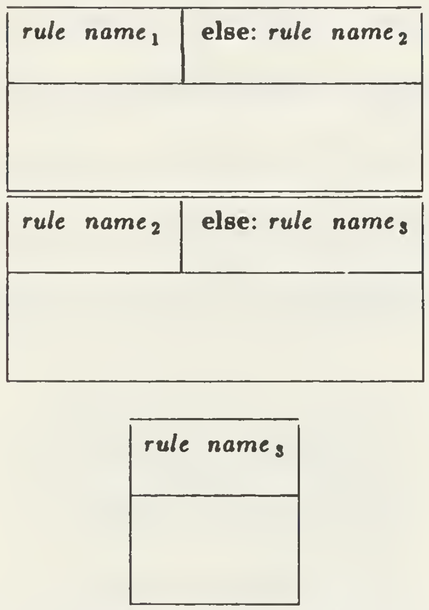
The two-dimensional Ω syntax, Ω_4, is based on the idea of *forms*. These can be thought of, and are displayed like, paper forms with fields that can be filled in with values. In particular, a relation is considered to be a blank form (i.e., a template), and each tuple in a relation is considered to be a filled out *instance* of that form.

Users can explicitly create or delete form instances, that is, add or delete tuples to or from relations, by selecting a form name (relation name) from a menu and filling in the fields of the form. This is a very natural mode of operation for offices and similar environments. To demonstrate this, we use a form-oriented terminology to describe the syntax of Ω_4.

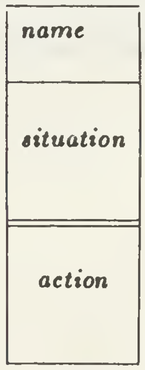
A *rule* is represented by a *rule window* labeled with the rule's name:



If the rule is part of a compound rule, then the right half of the rule's title chains to the alternative rule:



Each rule window is divided into two *frames,* an upper *situation frame* (the cause) and a lower *action frame* (the effect):



The situation frame is occupied by zero or more *condition panes,* which represent the presence or absence of form instances (tuples in relations). A condition pane has the format:

relation name
field–name$_1$: *field–pattern*$_1$
field–name$_2$: *field–pattern*$_2$
⋮
field–name$_n$: *field–pattern*$_n$

This kind of condition pane tests for the *presence* of the specified form instance (tuple).

The *field-patterns* can be either constants or variables. If they are constants then the pane will only match a form instance in which that field is filled in with that value. If the field-pattern is a variable, then the variable can be either unbound or bound. If it is unbound, then it will match any field-value, and will become bound to that field-value. If it is bound, then it matches only the value to which it is bound. Field-patterns can be left blank, which has the effect of filling them in with new, unique, unbound variables. Thus, blank field-patterns are considered "don't cares" since they match anything.

The *absence* of a form instance is indicated by appending the modifier '**(absent)**' to the form (relation) name:

relation name **(absent)**
field–name$_1$: *field–pattern*$_1$
field–name$_2$: *field–pattern*$_2$
⋮
field–name$_n$: *field–pattern*$_n$

Similarly, to test for the presence of a form instance, and delete the form instance when found, we append the modifier '**(delete)**':

relation name **(delete)**
field–name$_1$: *field–pattern*$_1$
field–name$_1$: *field–pattern*$_2$
$\vdots$
field–name$_n$: *field–pattern*$_n$

Finally, there is a special condition pane, called a *constraint pane,* which can appear in the situation frame:

Constraint
Boolean expression

There can be any number of constraint pane in the situation frame; they must all evaluate to **true** for the rule to apply.

The action frame is filled with a number of *transaction panes*. A transaction pane can have four formats: creation, deletion, procedure, or sequential process. A creation pane has the form:

relation name
field–name$_1$: *field–value*$_1$
field–name$_2$: *field–value*$_2$
$\vdots$
field–name$_n$: *field–value*$_n$

This calls for the creation of the specified form instance. The *field-values* are expressions used to compute the values for the fields.

A *deletion pane* calls for the deletion of the specified form instance:

relation name **(delete)**
$field\text{-}name_1$: $field\text{-}value_1$
$field\text{-}name_2$: $field\text{-}value_2$
$\vdots$
$field\text{-}name_n$: $field\text{-}value_n$

The third kind of action pane is a *procedure* or *call pane:*

relation name **(procedure)**
$field\text{-}name_1$: $field\text{-}value_1$
$field\text{-}name_2$: $field\text{-}value_2$
$\vdots$
$field\text{-}name_n$: $field\text{-}value_n$

This calls for a synchronous call of the specified relation. That is, the form instance is created, which requests some action to be performed. The rules containing the procedure call is not considered complete until the completion of the requested action is acknowledged via an 'Acknowledgement' form.

The last kind of transaction pane is a *sequential process pane:*

Sequence	
1:	$statement_1$
2:	$statement_2$
$\vdots$	$\vdots$
n	$statement_n$

The statements are executed in the order listed. They may be either the names of rule windows, or Ω rules in one of the linear forms (Ω_1, Ω_2, or Ω_3).

Finally, we need a means for manipulating groups of rules as a unit; this is the *rule-group window:*

Rules: *group name*
$rule - name_1$
$rule - name_2$
$\vdots$
$rule - name_n$

For example, a request to activate a rule group will activate all the rules named in that group.

A specification of the stack example in Ω_4 follows.

STACKS IN Ω_4

Rules: Stack Rules
Push Rule Pop Rule New Stack Rule Destroy Stack Rule

Push Rule	
Push Request (delete)	Stack Contents (delete)
From: A Item: X Stack: S	Stack: S List: Y
Acknowledgement	Stack Contents
To: A Concerning: S	Stack: S List: appending of X and Y

Pop Rule	
Pop Request (delete)	Stack Contents (delete)
From: A Stack: S	Stack: S List: X
Acknowledgement	Stack Contents
To: A Concerning: first of X	Stack: S List: rest of X

New Stack Rule	
New Stack Request (delete)	Available Object (delete)
From: A	ID: S
Acknowledgement	Stack Contents
To: A Concerning: S	Stack: S List: nil

Destroy Stack Rule	
Destroy Stack Request (delete)	Stack Contents (delete)
From: A Stack: S	Stack: S List: X
Acknowledgement	
To: A Concerning: X	

Dialog
27: Activate Stack Rules;
28: Define public Push Request to be create-only Push Request;
29: Define public Pop Request to be create-only Pop Request;
30: Define public Destroy Stack Request to be create-only Destroy Stack Request;
31: Define public New Stack Request to be create-only New Stack Request;
32: _

APPENDIX A: ABSTRACT SYNTAX OF Ω

$$\mathit{Session} \;=\; \mathit{statement-list}$$

$$\mathit{statement-list} \;=\; <\mathbf{statement{-}list};\ \mathit{statement}\ ;\ \mathit{statement-list?}>$$

$$\mathit{statement-list?} \;=\; \left\{\begin{array}{l} \mathbf{nil} \\ \mathit{statement-list} \end{array}\right\}$$

$$\mathit{statement} \;=\; \left\{\begin{array}{l} <\mathbf{else};\ \mathit{rule}\ ;\ \mathit{statement}> \\ <\mathbf{rule};\ \mathit{cause?}\ ;\ \mathit{effect?}> \end{array}\right\}$$

$$\mathit{compound-rule} \;=\; <\mathbf{cpd};\ \mathit{rule}\ ;\ \mathit{statement?}>$$

$$\mathit{statement} \;=\; \left\{\begin{array}{l} \mathbf{nil} \\ \quad \mathit{statement} \end{array}\right\}$$

$$\mathit{rule} \;=\; <\mathbf{rule};\ \mathit{cause}\ ;\ \mathit{effect?}>$$

$$\mathit{cause} \;=\; <\mathbf{cause};\ \mathit{condition}\ ;\ \mathit{cause?}>$$

$$\mathit{cause?} \;=\; \left\{\begin{array}{l} \mathbf{nil} \\ \mathit{cause} \end{array}\right\}$$

$$\mathit{condition} \;=\; \left\{\begin{array}{l} <\mathbf{present};\ \mathit{inquiry}> \\ <\mathbf{absent};\ \mathit{inquiry}> \\ <\mathbf{cancel};\ \mathit{inquiry}> \end{array}\right\}$$

$$\mathit{inquiry} \;=\; <\mathbf{inquiry};\ \mathit{primary}\ ;\ \mathit{tupl-pattern}>$$

$$\mathit{tupl-pattern} \;=\; \left\{\begin{array}{l} <\mathbf{tupl};\ \mathit{pattern}\ ;\ \mathit{tupl-pattern?}> \\ <\mathbf{arbtupl};\ \mathit{pattern}\ ;\ \mathit{pattern}> \end{array}\right\}$$

$$\mathit{tupl-pattern?} \;=\; \left\{\begin{array}{l} \mathbf{nil} \\ \mathit{tupl-pattern} \end{array}\right\}$$

$$\mathit{pattern} \;=\; \left\{\begin{array}{l} \mathit{free\text{-}variable} \\ \mathit{expression} \end{array}\right\}$$

$$\mathit{free\text{-}variable} \;=\; \left\{\begin{array}{l} \mathbf{nil} \\ <\mathbf{var};\ \mathit{string}> \end{array}\right\}$$

$$\mathit{effect?} \;=\; \left\{\begin{array}{l} \mathbf{nil} \\ \mathit{effect} \end{array}\right\}$$

$$\mathit{effect} \;=\; <\mathbf{effect};\ \mathit{transaction}\ ;\ \mathit{effect?}>$$

$$\mathit{transaction} \;=\; \left\{\begin{array}{l} <\mathbf{assert};\ \mathit{predication}> \\ <\mathbf{deny};\ \mathit{predication}> \\ \mathit{call} \\ \mathit{seq-block} \end{array}\right\}$$

predication = <**predication**; *primary* ; *arguments?* >

$$\textit{arguments?} \;=\; \left\{\begin{array}{l}\textbf{nil}\\ \textit{arguments}\end{array}\right\}$$

arguments = <**arguments**; *expression* ; *arguments?* >

call = <**call**; *primary* ; *arguments?* >

seq−block = <**seq−block**; *statement−list* >

$$\textit{expression} \;=\; \left\{\begin{array}{l}<\textit{binop}\;;\;\textit{expression}\;;\;\textit{expression}>\\ <\textit{unop}\;;\;\textit{expression}>\\ \textit{primary}\end{array}\right\}$$

$$\textit{binop} \;=\; \left\{\begin{array}{l}\textbf{or}\\ \textbf{and}\\ \textbf{eq}\\ \textbf{ne}\\ \textbf{lt}\\ \textbf{gt}\\ \textbf{le}\\ \textbf{ge}\\ \textbf{sum}\\ \textbf{dif}\\ \textbf{prd}\\ \textbf{quo}\\ \textbf{mod}\end{array}\right\}$$

$$\textit{unop} \;=\; \left\{\begin{array}{l}\textbf{not}\\ \textbf{neg}\end{array}\right\}$$

$$\textit{primary} \;=\; \left\{\begin{array}{l}<\textbf{con};\;\textit{value}>\\ <\textbf{selfref};\;\textit{var}>\\ <\textbf{var};\;\textit{string}>\\ <\textbf{apply};\;\textit{var}\;;\;\textit{arguments}>\\ <\textbf{eval};\;\textit{expression}\;;\;\textit{expression}>\\ \textit{list}\\ \textit{call}\\ <\textbf{rule-den};\;\textit{statement-list}>\end{array}\right\}$$

$$\textit{list} \;=\; \left\{\begin{array}{l}\textit{listing}\\ <\textbf{cons};\;\textit{expression}\;;\;\textit{expression}>\end{array}\right\}$$

$$\textit{listing} \;=\; \left\{\begin{array}{l}\textbf{nil}\\ <\textbf{list};\textit{expression}\;;\;\textit{listing}>\end{array}\right\}$$

APPENDIX B: SYNTAX OF Ω_1

$$\mathit{Session} \;=\; \mathit{statement-list}\ .$$

$$\mathit{statement-list} \;=\; \mathit{statement}\ ;\ \cdots$$

$$\mathit{statement} \;=\; \left\{\begin{matrix} \mathit{rule}\ \ \text{else}\ \mathit{statement} \\ [\mathit{cause}\ \Rightarrow]\ \mathit{effect} \end{matrix}\right\}$$

$$\mathit{compound-rule} \;=\; \mathit{rule}\ [\ \text{else}\ \mathit{statement}\ ]$$

$$\mathit{rule} \;=\; \mathit{cause}\ \Rightarrow\ \mathit{effect}$$

$$\mathit{cause} \;=\; [\mathbf{if}]\ \mathit{condition}\ ,\ \cdots$$

$$\mathit{condition} \;=\; \begin{bmatrix} * \\ \neg \end{bmatrix}\ \mathit{inquiry}$$

$$\mathit{inquiry} \;=\; \mathit{primary}\ (\ \mathit{tupl-pattern}\ )$$

$$\mathit{tupl-pattern} \;=\; \left\{\begin{matrix} \mathit{pattern}\ ,\ \cdots \\ \mathit{pattern}\ :\ \mathit{pattern} \end{matrix}\right\}$$

$$\mathit{pattern} \;=\; \left\{\begin{matrix} \mathit{free\text{-}variable} \\ \mathit{expression} \end{matrix}\right\}$$

$$\mathit{free\text{-}variable} \;=\; \left\{\begin{matrix} - \\ \mathit{variable} \end{matrix}\right\}$$

$$\mathit{effect} \;=\; [\mathit{transaction}\ ,\ \cdots]$$

$$\mathit{transaction} \;=\; \left\{\begin{matrix} \mathit{assertion} \\ \mathit{denial} \\ \mathit{call} \\ \mathit{seq-block} \end{matrix}\right\}$$

$$\mathit{assertion} \;=\; \mathit{predication}$$

$$\mathit{denial} \;=\; \neg\ \mathit{predication}$$

$$\mathit{predication} \;=\; \mathit{primary}\ (\ \mathit{arguments}\ )$$

$$\mathit{call} \;=\; \mathit{primary}\ \{\ \mathit{arguments}\ \}$$

$$\mathit{arguments} \;=\; [\mathit{expression}\ ,\ \cdots]$$

$$\mathit{seq-block} \;=\; \{\ \mathit{statement-list}\ \}$$

$$\mathit{expression} \;=\; [\mathit{expression}\ \bigvee\ ]\ \mathit{conjunction}$$

$$\mathit{conjunction} \;=\; [\mathit{conjunction}\ \bigwedge\ ]\ [\neg]\ \mathit{relation}$$

$$\mathit{relation} \;=\; [\mathit{simplex}\ \ \mathit{relator}]\ \mathit{simplex}$$

$$\mathit{relator} \;=\; \{= \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant\}$$

$$\mathit{simplex} \;=\; [\mathit{simplex}\ \{+ \mid -\}]\ \mathit{term}$$

$$\mathit{term} \;=\; [\mathit{term}\ \{* \mid / \mid \%\}]\ \mathit{factor}$$

$$\mathit{factor} \;=\; \begin{bmatrix} + \\ - \end{bmatrix} \mathit{primary}$$

$$\mathit{primary} \;=\; \mathit{primitive}\ [:\ \mathit{primary}]$$

$$\mathit{primitive} \;=\; \left\{\begin{array}{l} \mathit{constant} \\ [@\,]\ \mathit{variable} \\ \mathit{primitive}\ [\ \mathit{arguments}\ ] \\ (\ \mathit{expression}\ ) \\ \left[\ \left\{\begin{array}{l} \mathit{expression}\ ,\ \cdots \\ \mathit{expression}\ :\ \mathit{expression} \end{array}\right\}\ \right] \\ \mathit{call} \\ \mathit{rule-denotation} \end{array}\right\}$$

$$\mathit{constant} \;=\; \left\{\begin{array}{l} \mathit{digit}^{+} \\ '' \left\{\begin{array}{c} \mathit{char} \\ '''' \end{array}\right\}^{*} '' \\ \mathbf{nil} \end{array}\right\}$$

$$\mathit{rule-denotation} \;=\; \ll \{\ \mathit{compound-rule}\ .\ \}^{*} \gg$$

APPENDIX C: SYNTAX OF Ω_2

$\textit{Session} \;=\; \textit{statement}-\textit{list}\ .$

$\textit{statement}-\textit{list} \;=\; \textit{statement}\ ;\ \cdots$

$$\textit{statement} \;=\; \left\{\begin{matrix}\textit{rule}\ \ \textbf{else}\ \textit{statement}\\ {[}\textit{cause}\ \ \textbf{then}]\ \textit{effect}\end{matrix}\right\}$$

$\textit{compound}-\textit{rule} \;=\; \textit{rule}\ [\ \textbf{else}\ \textit{statement}\ ]$

$\textit{rule} \;=\; \textit{cause}\ \ \textbf{then}\ \textit{effect}$

$\textit{cause} \;=\; \textbf{When}\ \textit{condition}\ \ \textbf{and}\ \ \cdots$

$\textit{condition} \;=\; [\textbf{given}]\ \textit{inquiry}$

$$\textit{inquiry} \;=\; [\textit{noise}-\textit{word}]\ \textit{primary}-\textit{pattern}\ \left\{\begin{matrix}\textit{word}\ [\textbf{not}]\\ \textbf{does}\ [\textbf{not}]\ \textit{word}\end{matrix}\right\}\ \textit{word}^{*}\ \ [\textit{tupl}-\textit{pattern}]$$

$$\textit{primary}-\textit{pattern} \;=\; \left\{\begin{matrix}\textit{free-variable}\\ \textit{primary}\end{matrix}\right\}$$

$$\textit{free-variable} \;=\; \left\{\begin{matrix}\textbf{anything}\\ \textit{variable}\end{matrix}\right\}$$

$$\textit{tupl}-\textit{pattern} \;=\; \left\{\begin{matrix}\textit{pattern}\ \ \{\textit{word}^{+}\ \textit{pattern}\}^{*}\\ \textbf{arbitrary}\ \textit{pattern}\end{matrix}\right\}$$

$$\textit{pattern} \;=\; \left\{\begin{matrix}\textit{variable}\\ \textit{expression}\end{matrix}\right\}$$

$\textit{noise}-\textit{word} \;=\; \{\ \textbf{a}\ \mid\ \textbf{an}\ \mid\ \textbf{the}\ \}$

$\textit{effect} \;=\; [\textit{transaction}\ \ \textbf{and}\ \ \cdots]$

$$\textit{transaction} \;=\; \left\{\begin{matrix}\textit{predication}\\ \textit{call}\\ \textit{seq}-\textit{block}\end{matrix}\right\}$$

$$\textit{predication} \;=\; [\textit{noise}-\textit{word}]\ \textit{primary}\ \left\{\begin{matrix}\textit{word}\ [\textbf{not}]\\ \textbf{does}\ [\textbf{not}]\ \textit{word}\end{matrix}\right\}\ \textit{word}^{*}\ \ [\textit{arguments}]$$

$\textit{call} \;=\; [\textit{noise}-\textit{word}]\ \textit{word}^{+}\ [\textit{arguments}]$

$\textit{arguments} \;=\; \textit{expression}\ \ \{\textit{word}^{+}\ \textit{expression}\}^{*}$

$\textit{seq}-\textit{block} \;=\; \textbf{begin}\ \textit{statement}-\textit{list}\ \ \textbf{end}$

$\textit{expression} \;=\; [\textit{expression}\ \ \textbf{or}]\ \textit{conjunction}$

$\textit{conjunction} \;=\; [\textit{conjunction}\ \ \&\ ]\ [\textbf{not}]\ \textit{relation}$

$$relation \;=\; [simplex \;\; relator\,]\; simplex$$

$$relator \;=\; \{= \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant\}$$

$$simplex \;=\; [simplex \;\; \{+ \mid -\}]\; term$$

$$term \;=\; [term \;\; \{* \mid / \mid \%\}]\; factor$$

$$factor \;=\; \begin{bmatrix} + \\ - \end{bmatrix} primary$$

$$primary \;=\; primitive \;\; [:\; primary\,]$$

$$primitive \;=\; \left\{ \begin{array}{l} constant \\ [\mathbf{own}]\; variable \\ word^{+}\; \mathbf{of}\; arguments \\ (\; expression \;\;) \\ \left[\left\{ \begin{array}{l} expression \;\; , \;\; \cdots \\ expression \;\; \mathbf{append}\; expression \end{array} \right\} \right] \\ call \\ rule-denotation \end{array} \right\}$$

$$constant \;=\; \left\{ \begin{array}{l} digit^{+} \\ {''} \left\{ \begin{array}{c} char \\ {''''} \end{array} \right\}^{*} \;\; {''} \\ \mathbf{nil} \end{array} \right\}$$

$$rule-denotation \;=\; \mathbf{rules}\; \{\; compound-rule \;\; .\; \}^{*} \;\; \mathbf{end\;rules}$$

APPENDIX D: SYNTAX OF Ω_3

Session = *statement − list* .

statement − list = *statement* ; · · ·

$$\textit{statement} \;=\; \begin{Bmatrix} \textit{rule}\ \textbf{else}\ \textit{statement} \\ [\textit{cause}\ \textbf{then}]\ \textit{effect} \end{Bmatrix}$$

compound − rule = *rule* [**else** *statement*]

rule = *cause* **then** *effect*

cause = **When** *condition* **and** · · ·

condition = [**given**] *inquiry*

inquiry = *noun − phrase* *verb − phrase*

$$\textit{noun} - \textit{phrase} \;=\; \left\{\begin{array}{l} \textit{determ}\ \textit{noun}\ \textit{arguments}\ [\textbf{which}\ \textit{verb} - \textit{phrase}] \\ \begin{Bmatrix} \textbf{that} \\ \textbf{something} \end{Bmatrix} \textbf{which}\ \textit{verb} - \textit{phrase} \\ \textit{expression} \end{array}\right\}$$

$$\textit{verb} - \textit{phrase} \;=\; \left\{\begin{array}{l} \textit{verb}\ [\textbf{not}]\ \textit{arguments} \\ \textbf{does}\ [\textbf{not}]\ \textit{verb}\ \textit{arguments} \\ \textbf{is}\ [\textbf{not}] \begin{Bmatrix} \textit{noun} - \textit{phrase} \\ \textit{adj} - \textit{phrase} \end{Bmatrix} \end{array}\right\}$$

adj − phrase = *adjective* *arguments*

$$\textit{arguments} \;=\; \begin{Bmatrix} \textit{prep} - \textit{phrase}^{*} \\ \textbf{arbitrary}\ \textit{variable} \end{Bmatrix}$$

prep − phrase = [*preposition*] *noun − phrase*

effect = [*transaction* **and** · · ·]

$$\textit{transaction} \;=\; \begin{Bmatrix} \textit{declaration} \\ \textit{call} \\ \textit{seq} - \textit{block} \end{Bmatrix}$$

declaration = *noun − phrase* *verb − phrase*

call = *verb* *arguments*

seq − block = **begin** *statement − list* **end**

expression = [*expression* or] *conjunction*

$$\mathit{conjunction} \;=\; [\mathit{conjunction}\ \&]\ [\mathbf{not}]\ \mathit{relation}$$

$$\mathit{relation} \;=\; [\mathit{simplex}\ \ \mathit{relator}]\ \mathit{simplex}$$

$$\mathit{relator} \;=\; \{= \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant\}$$

$$\mathit{simplex} \;=\; [\mathit{simplex}\ \ \{+ \mid -\}]\ \mathit{term}$$

$$\mathit{term} \;=\; [\mathit{term}\ \ \{* \mid / \mid \%\}]\ \mathit{factor}$$

$$\mathit{factor} \;=\; \begin{bmatrix} + \\ - \end{bmatrix} \mathit{primary}$$

$$\mathit{primary} \;=\; \mathit{primitive}\ \ [:\ \mathit{primary}]$$

$$\mathit{primitive} \;=\; \left\{\begin{array}{l} \mathit{constant} \\ [\mathbf{own}]\ \mathit{variable} \\ \mathit{primitive}\ \ [\mathit{arguments}] \\ (\ \mathit{expression}\ \ ) \\ \left[\left\{\begin{array}{l} \mathit{expression}\ \ ,\ \ \cdots \\ \mathit{expression}\ \ \mathbf{append}\ \mathit{expression} \end{array}\right\}\right] \\ \mathit{call} \\ \mathit{rule-denotation} \end{array}\right\}$$

$$\mathit{constant} \;=\; \left\{\begin{array}{l} \mathit{digit}^{+} \\ \text{''}\left\{\begin{array}{c} \mathit{char} \\ \text{''''} \end{array}\right\}^{*}\ \ \text{''} \\ \mathbf{nil} \end{array}\right\}$$

$$\mathit{rule-denotation} \;=\; [\mathbf{the}]\ \mathbf{rules}\ \{\ \mathit{compound-rule}\ \ .\ \}^{*}\ \mathbf{end\ rules}$$

$$\mathit{determ} \;=\; \left\{\begin{array}{l} \mathbf{a} \\ \mathbf{an} \\ \mathbf{the} \end{array}\right\}$$

$$\mathit{noun} \;=\; \mathit{word}^{+}$$

$$\mathit{verb} \;=\; \mathit{word}^{+}$$

$$\mathit{adjective} \;=\; \mathit{word}^{+}$$

$$\mathit{preposition} \;=\; \mathit{word}^{+}$$

APPENDIX E: SYNTAX OF Ω_4

Session =

Dialog
statement − list

statement − list =

$$\begin{array}{l} 1\colon \Omega_3-statement_1 \\ 2\colon \Omega_3-statement_2 \\ \quad\vdots \\ n\colon \Omega_3-statement_n \end{array}$$

compound − rule =

$name_1$	**else:** $name_2$
cause	
effect	

or

name
cause
effect

cause = $condition^+$

condition =

name [(*modifier*)]
$name_1$: $pattern_1$ $\vdots$ $name_n$: $pattern_n$

(If the field names are **first** and **rest**, then the pattern matches the first and rest of an arbitrary tuple.)

$$modifier = \left\{ \begin{array}{l} \textbf{delete} \\ \textbf{absent} \end{array} \right\}$$

$$pattern = \left\{ \begin{array}{l} variable \\ \Omega_3-expression \end{array} \right\}$$

$$effect = transaction^*$$

$$transaction = \left\{ \begin{array}{l} nonsequential \\ sequential \end{array} \right\}$$

nonsequential =

name [(*kind*)]
$name_1$: $\Omega_3-expression_1$ $\vdots$ $name_n$: $\Omega_3-expression_n$

$$kind = \left\{ \begin{array}{l} \textbf{delete} \\ \textbf{procedure} \end{array} \right\}$$

sequential =

Sequence
statement − list

rule − denotation =

Rules: *name*
$name_1$ $\vdots$ $name_n$

APPENDIX F: INCOMPATIBILITIES WITH MCARTHUR PROTOTYPE

There are a number of minor syntactic incompatibilities between the dialect of Ω_1 implemented by the McArthur prototype and that described in this report.

1. To simplify parsing, the keyword **if** is required on all rules with a nonnull cause.
2. The lexical representation of '$\Rightarrow$' is '->', and strings are surrounded by the ASCII double quote symbol.
3. Additional degenerate forms of rules, such as '**if** *cause* $\Rightarrow$', are permitted.
4. A user can enter multiple *sessions,* each terminated by a period. The period calls for the execution of all statements in that session. The semicolon statement termination does not cause execution. Rather, the statements are saved until the next period.
5. The McArthur prototype does not distinguish between statements and compound rules. The result is that it is possible to activate rules with an empty cause part.
6. Arbitrary expressions are permitted as transactions.
7. The object-oriented language Ω is augmented with an applicative sublanguage. To support this, statements include function declarations of the form:

   $$\textbf{function}\ \textit{variable}\ [\ \textit{formals}\ ] : \textit{compound}-\textit{expression}$$

   where

   $$\textit{formals} = \begin{Bmatrix} \textit{variable}\ , \cdots \\ \textit{variable}\ : \textit{variable} \end{Bmatrix}$$

   and

   $$\begin{aligned} \textit{compound}-\textit{expression} &= \textit{cond}-\textit{expression}\ \textbf{else}\ \cdots \\ \textit{cond}-\textit{expression} &= [\textbf{if}\ \textit{expression} \Rightarrow]\ \textit{expression} \end{aligned}$$

8. Mutually recursive functions are declared by means of a "forward" declaration:

   function f [$\cdots$]: **nil**;
   function g [$\cdots$]: $\cdots$ f $\cdots$;
   function f [$\cdots$]: $\cdots$ g $\cdots$;

   This ensures that f is bound before it's used in g, and that g is bound before it's used in f.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 2

Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943 2

Office of Research Administration
Code 012A
Naval Postgraduate School
Monterey, CA 93943 1

Chairman, Code 52Ml
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 40

Associate Professor Bruce J. MacLennan
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 12

Dr. Robert Grafton
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217-5000 1

Dr. David W. Mizell
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106 1

Professor Jack M. Wozencraft, 62Wz
Department of Elec & Computer Engr.
Naval Postgraduate School
Monterey, CA 93943 1

Professof Rudolf Bayer
Institüt für Informatik
Technische Universität
Postfach 202420
D-8000 Munchen 2
West Germany 1

Dr. Robert M. Balzer USC Information Sciences Inst. 4676 Admiralty Way Suite 10001 Marina del Rey, CA 90291	1
Mr. Ronald E. Joy Honeywell, Inc. Computer Sciences Center 10701 Lyndale Avenue South Bloomington, MI 55402	1
Mr. Ron Laborde INMOS Whitefriars Lewins Mead Bristol Great Britain	1
Mr. Lynwood Sutton Code 424, Building 600 Naval Ocean Systems Center San Diego, CA 92152	1
Mr. David Lefkovitz 310 Cynwyd Road Bala Cynwyd, PA 19004	1
Mr. Jeffrey Dean Advanced Information and Decision Systems 201 San Antonio Circle, Suite 286 Mountain View, CA 94040	1
Mr. Jack Fried Mail Station D01/31T Grumman, Aerospace Corporation Bethpage, NY 11714	1
Mr. Dennis Hall 2 Ivy Drive Orinda, CA 94563	1
Mr. A. Dain Samples Computer Science Division -EECS University of California at Berkley Berkley, CA 94720	1
Professor S. Ceri Laboratorio Di Calcolatori Departimento di Elettronica Politecnico di Milano 20133 - Milano Italy	1